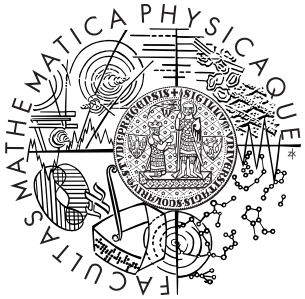
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

# BAKALÁŘSKÁ PRÁCE




Václav Remeš

## Point based editor

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán
Studijní program: informatika, programování

2008

Děkuji mým rodičům a snoubence za veškerou podporu, pomoc a účast,
mému vedoucímu za všechny poskytnuté rady
a všem, kteří mi pomáhali neztratit rozum během psaní této práce.

Prohlašuji, že jsem svou bakalářskou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 7.8.2008 Václav Remeš

# Contents

Název práce: Point based editor
Autor: Václav Remeš
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. Josef Pelikán
e-mail vedoucího: josef.pelikan@mff.cuni.cz

Abstrakt: Cílem této práce je imlementace interaktivního 3D editoru schopného pracovat s neorgazinovanými mračny bodů. Netradiční grafická primitiva potřebují netradiční editační nástroje, autor by měl navrhnout a implementovat několik instrumentá a ověřit jejich chování na reálných editačních úlohách.

Částečné problémy k vyřešení: účinná representace dat a robustnost, počítání normálových vektorů, dynamický level-of-detail, stejnoměrná hustota bodů, atd.

Aplikace by měla být dobře okomentovaná a zdokumentované - podle MFF UK konvencí.
Klíčová slova: point clouds editor

Title: Point-based editor
Author: Václav Remeš
Department: Kabinet software a výuky informatiky
Supervisor: RNDr. Josef Pelikán
Supervisor's e-mail address: josef.pelikan@mff.cuni.cz

Abstract: Goal of this thesis is to implement interactive 3D editor able to work with unorganized point-clouds. Nontraditional graphics primitive needs nontraditional editing tools, author should propose and implement several instruments and verify their behavior in real editing tasks.

Particular problems to be solved: efficient data representation and persistence, normal vector computation, dynamic level-of-detail, isotropic point density, etc.
Keywords: point clouds editor

# Chapter 1

# Introduction

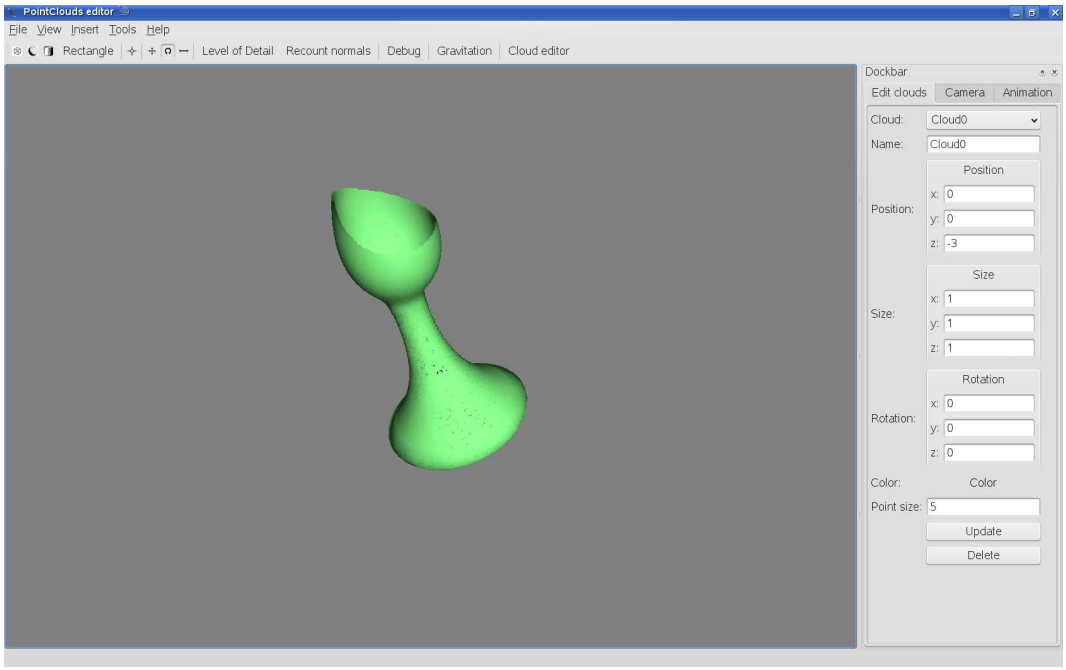
## 1.1 About point-based graphics

Point-based graphics (as described for example in [1])is a very specific and unorthodox way of displaying 3D objects in computer environment. The two typical attitudes involve either volume-based represantion (e.g. CSG trees, used in ray-tracing) or triangular mesh representation, which is used for example in most (probably all) the 3D games. These two attitudes are well-known, in case of the triangular representation even very time-efficient, and when rendered produce high-quality images.

The point-based attitude to displaying objects is based on the idea of representing each object as a bunch of points (called a point cloud) that, when given enough of them, carry all the information about the shape of the object and even look as a continuous surface (however, for this, the numbers of the points have to be quite large - the numbers have to be in ranges about $10^5$ - $10^6$ for the point cloud to look continuously considering that every point uses 1 pixel (the continuousness will be discussed later).

The question then is - why another representation, that even has problems with shading, continuousness, etc.? It's because there are some operations, that can be performed extremely simply on point clouds and for example - when you do a 3D scan, the result is just a point cloud which of course can be transformed to, for example, nurbs, but it could be useful to let the point cloud be in the state it is or edit it a bit before the transformation.

Another example of use are simulation of physical processes of a mechanical continuum. When the points of the point cloud are looked on as physical particles (probably molecules or some dribblets) that can interact (e.g. gravitationaly, or just mechanicaly coliding) with their environment, many interresting simulations can be performed, because since there is quite a number of points, the points can behave almost like liquids.




## 1.2 The goal of the work

The goal of the program was to create a functioning editor for point based graphics that would have decent deformation tools, allow the user to save the scene and later load it from a file, and to take handle of the normal vectors and level of detail - i.e. displaying the point clouds in such a way that they can look almost like made of a continuous surface.

This work then tries to bring information about the process of programming the editor, give theoretical background to the algorithms, data structures and mathematics used, etc. It covers the area of some space-splitting tree structures, efficient space search for nearest neighbours and makes a quick insight into principal component analysis, which is used to get the basic data about the point clouds such as the normal vectors of the actual points, adjust level of detail, etc.

## 1.3 Main issues

When point-based graphics is used, the first problem, that arises (apart from how to display them...) is how to store them - I had to choose a data structure which makes it possible to display the points effectively, and that also makes the deformation as easy

to apply as possible and which also is sophisticated enough to grant me a possibility to search through the points with a low time complexity.

Secondly, when you display the points, a problem is, that the points are very fine, but the computer doesn't know how to shade them by itself, since it doesn't know the normal vectors of the points, which are needed to do the shading job. The points of the clouds then have to be analyzed in relations with each other and the editor has to be able to compute the normal vectors from the organization of the neighbourhood of each point.

Another problem is, that if you start deforming the point cloud, then even if the point cloud had the points spread ideally and looked continuous, after most of the deformations it can end up full of holes, thin and not continuous or on the other hand it can look alright, but the points will be very dense and ineffectively used and some of them can be deleted without affecting the image.

The remaining issues that followed the project were quite minor - how to deform the cloud, how to store the point cloud in a file, the tools for deformation, etc.

# Chapter 2

# Theoretical background - data structures for storing the points

Since the theme that the work most focuses on are point clouds - objects that contain tens or hundreds of thousands of points, storing the points only simply in arrays wouldn't provide many efficient ways of working with them. Although I have the points stored also in arrays (will be covered further in this section), they are mainly used for deforming the cloud, for the points can thus be lineary walked through and no complicated tree needs to be affected (nevertheless, the deformation surely would break the trees invariants).

In the program I implemented two tree-based data structures for storing the points (and one structure storing the points in an array, just for comparison of how the tree structures make the computing less time and processor consuming), which I will describe here.

Also some auxiliary data structures had to be implemented to help with searching through the trees and finding the n nearest neighbours of a point in space (in which case a heap is used) and finding all points of the point cloud in a given distance from a point in space (in which occasion the gum array is used).

## 2.1 Storing the points

For storing the points I use two data structures which store the points redundantly (they both store the same values) - a simple array of points which is there mainly for displaying the point clouds and a more sophisticated data structure to help me with searching through the points and all the other computational-intensive stuff (I have two usable implementations - a KD-tree and a bucket KD-tree).

I made the decision of having two redundant data structures after a careful consideration - I definitely needed some sort of a robust, probably tree-like, data structure for searching for the nearest points, getting all the points in an area and so on, but this structure would be very difficult to give to the opengl function for drawing, which takes only arrays as its parameter (to work efficiently). Of course, I could have simply implemented a method for transforming the tree into an array of points (which I have done nonetheless) and call the function every time the point cloud would be displayed, but this would consume too much processor time and would be inefficient. Moreover, the simple array of points allows me to apply most of the deformations in a single for-cycle.

To sum it, I have two equal representations of the point cloud with only one of them carrying the up-to-date information about the whole cloud. When I need the other structure, I have to call the transformation functions. This approach works quite fine, since the transformation is usually done before some sort of a time-consuming operation is performed and it is quite fast in comparison to all the more complex operations.

The "more sophisticated data structure" for searching through the point cloud is an object of one of the classes derived from the CTree abstract class, which defines the interface. This allowed me to have several implementations for the "searching data structure". Actually there are now three (two of them usable in normal life) implementations of this class - the CDebugT, CKDTree and the CBucketKDTree. The first one being there just to see the effectiveness of searching through the points when stored only in an array, the other two being the implementation of a KD-Tree and a bucket KD-Tree.

## 2.2 The parental structure defining the interface

To be able to switch between several data structures and compare their effectiveness in solving the main problems of the editor (counting the normal vectors and level of detail) I had to define an interface, which would be inherited by all the data structures.

The CTree class is the common parent of all the data structures used to store the points of a point cloud. It is an abstract class and hence cannot be instantiated, but it defines the interface each of the structures has to implement. Since they are all inheriting from this class, I could simply have the tree variable of the CGLPointCloud be a pointer to the CTree class and just assign into it objects of various of the implementing classes and compare them between each other.

The functions that the CTree class defines are inserting, searching for nearest neighbours of a point (either for a given number or for all points in a given distance), transformation between the tree-based representation and arrays, functions for arranging

normal vectors and some debug functions for testing.

## 2.3 Comparison to the tree-based data structures - an array

For the purposes of debuging and comparison, I had implemented the simpliest data structure imaginable (the class CDebugT) for storing and searching through the points of the cloud. This class implements a data structure that could (and even should) be easily overlooked as it is inefficient and time(processor)-consuming in all the tasks it can perform. It is a simple, unsorted array of points and I put it in the program because of only one reason - to have something to compare the other data structures with. It has only one advantage - CDebugT's code is so short and simple that it took me no time to write it and there isn't even a slightest probability, that one could make a mistake in it.

**Efficiency of the structure's operations** Since the points in this structure are stored using the gum array (as described later), the insert operation has an amortized complexity of $O(n)$, although, in the worst case, inserting one element can take $O(n)$ operations.

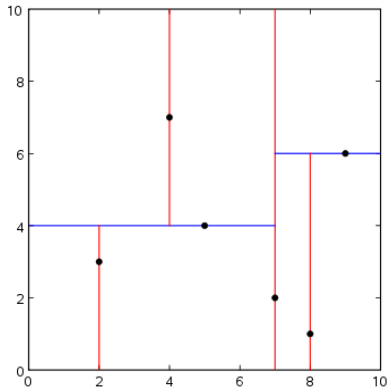
Both searching for nearest n neighbours of a point and for all the neighbours in a given distance from the starting point search through the whole array for every point, so their time complexity is $O(n^2)$.

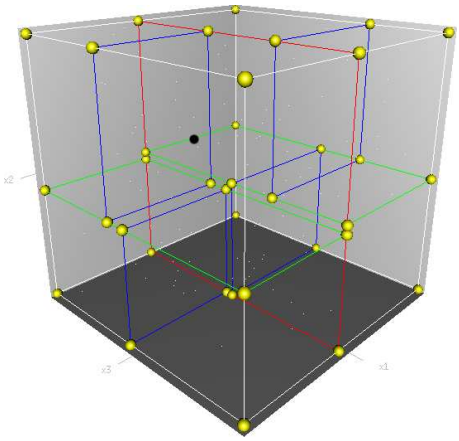
## 2.4 KD-Tree

A KD-tree (in short a k-dimensional tree as decribed in [3]; in the program it is implemented by the CKDTree class) is a space partitioning data structure for organizing points in k-dimensional space. KD-tree is a special type of a binary tree. Each node defines a splitting plane that divides space in one of the dimensions (the splitting dimension is the same for all nodes in one level of the tree and changes evenly in each level of the KD-tree).

One point is stored in every node of the tree and its coordinates define the splitting plane. Left son of each node holds a subtree of points that have a lesser value of the splitting plane's defining coordinate and right son holds a subtree of points that have the leading coordinate greater or equal.

The following two pictures were downloaded from wikipedia.org (they are licensed under gpl, so I hope it's fine if I put them here). The first one shows an example 2-dimensional kd-tree, the second one shows a 3D kd-tree.



The CKDTree class implements a 3-dimensional unbalanced KD-tree. In the program, I construct a KD-tree from an array of points by going through the array and inserting the point into the tree using the Insert operation. The Insert operation of a KD-tree works as the operation for searching for a certain point - it starts in the root node and continues into the left/right son depending on the value of the leading coordinate of the node it is currently in (in my implementation, the coordinates cycle evenly - 1., 2., 3., 1., 2.,... etc.). If the value of the leading coordinate of the inserted point is lesser, it goes into the left son, if it is greater or equal, it goes into the right son. When the searching ends in a leaf node of the tree, it creates a new node containing the inserted data and sets it as the appropriate son of the previously leaf node (the newly inserted node then becomes a new leaf of the tree).

It is certain, that the tree could degenerate into an array, when given badly sorted data for inserting, but when I experimented with inserting randomly-generated points (which is reasonable, since the points of the point cloud are randomly generated), the tree of about 10 000 vertexes had a height of 35 - 40 - on average, the height is $O(\lg n)$ where n is the number of points in the tree. Given this knowledge, it is easy to see, that the insert operation, since it goes directly from the root of the tree to a leaf, has time complexity also $O(\lg n)$.

In each node of the tree I remember not only the position of the stored point but also the normal vector belonging to it. This is needed in the operation for transforming the tree into the two arrays - one holding the points and the other storing their corre-

spondent normal vectors. This operation walks through all the points of the tree and puts the data in the preallocated arrays.

### 2.4.1 The searching operations.

Before we can continue to searching for n nearest neighbours of a point and for all neighbours in a given distance from a starting point, searching for a nearest point in the KD-tree needs to be described, for the remaining two algorithms use the same idea. The simple search for a nearest point to a given location starts in the root node of the tree, and adds it as the first candidate for nearest neighbour (which is then tested in each node and replaced if the node's point is nearer). Then in compares the location's coordinates to the leading coordinate of the node. If it is lower, it continues the same test (with the appropriate coordinate though) in the left son, otherwise in the right son, until it reaches a leaf node. However, the search algorithm doesn't end now, for when considered the splitting of space the KD-tree does, it could have happened, that the nearest neighbour could be in another branch of the tree, that the algorithm could have skipped.

So, when all the way down in the leaf, the search algorithm then has to go back up and in every node test, if the nearest neighbour could be in the other (not previously tested) branch of the tree (this is tested simply by comparing the difference of the values of the leading coordinate with the distance to the current candidate). There starts a recursion and follows like the first stage of the algorithm. We can see, that in the worst case, the complexity of the search algorithm could be $O(n)$, where n is the number of nodes in the tree, but this doesn't happen very often and on average, the complexity is once again $O(\lg n)$.

**Search for n nearest neighbours of a point** begins and runs similarly to the algorithm for searching for the nearest neighbour. The difference is that instead of carrying the current best result with it through the tree, the algorithm holds the n nearest found neighbours in a data structure that can quickly return the distance of the furthest element and replace it with a better one (this is implemented as a heap in the class CNearestNResult). So, in each node of the tree visited, the algorithm tests whether the node's point can improve the result - i.e. if it is nearer than the furthest currently found neighbour and if it is, the algorithm inserts the node's point in its place (and rearranges the heap to fulfill the heap invariants). The algorithm then continues exactly like the search for nearest neighbour, i.e. when the leaf is reached, the algorithm goes up and tests all the branches that weren't accessed and tests if they can contain a point that could be near enough. The algorithm ends when there are no branches that

could contain an improving point left.

**The algorithm for searching for all neighbours in a given distance** is much more simple than the one searching for nearest n neighbours. Since the starting point and the distance that is searched are given at the start of the algorithm, the algorithm starts in the root node and goes recursively through all the branches that contain the area searched. The test if the tested area lies within the branch of the tree is easy - since the searched area is spherical and we know its center and radius, the only thing that needs to be tested is the distance of the leading coordinates - if the distance is lesser than the radius, than the branch is visited and processed further.

All the points that lie within the searched area are stored using a gum array implemented in the CNearestDResult class and returned in the end. The algorithm ends when it visits all branches of the tree that contain the searched area.

## 2.5 Bucket KD-Tree

A bucket KD-tree is a variation of a KD-tree that uses the idea of splitting planes in each nodes and puts it together with using the buckets. A bucket stores a bunch of points together in a single node. Apart from using the buckets, the main differences between "normal" KD-trees and Bucket KD-trees is that all the points of the bucket KD-tree are stored in leaves and that the bucket KD-tree has a different insert operation (and also the delete operation but since I don't delete the points of the cloud in a conventional way, the delete operation wasn't needed to be implemented).

My implementation of the bucket KD-tree, the CBucketKDTree, is again 3-dimensional and unbalanced. Since the inner (non-leaf) nodes of the tree store no points, I had to add a member variable of type glVertex called split, which remembers the coordinates of the splitting plane.

The insert operation first goes through the tree as if searching if the inserted point is present and finds the bucket in which the point should be placed and tries to insert it. If there is enough space in the bucket, the insert operation terminates. However, if the bucket is full it has to be split. This is done by sorting all the points in the bucket by the values of the coordinate that is the leading coordinate in the actual level of the bucket KD-tree, then splitting them into two halves and creating two new leaf nodes with their buckets being filled with the points of either of the halves. The old leaf node's bucket is then deleted and the two new nodes are added as its sons.

This implementation grants me the certainty, that each node either has both of its sons or is a leaf node with a bucket. This knowledge is heavily used in the code while

going through the tree.

### 2.5.1 Searching through the bucket KD-tree

The searching for neighbours in a bucket KD-tree is pretty much similar to searching through a normal KD-tree. The only difference is, that since all the points are stored in the leafs in buckets, no testing needs to be done all the way down to the leaves and in every leaf several points are tested together instead of testing one point a time.

# Chapter 3

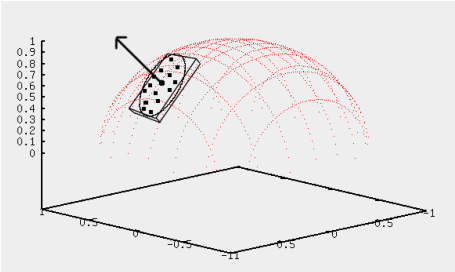
# Solutions to general problems of the work

As I said, the main problems were with shading the point clouds (this is where the counting of normal vectors comes in) and with level of detail (or, better said, with thickening the points, for reducing the number of points wasn't implemented due to its contraproductiveness - whenever a point would be removed from the point cloud, the program would loose valuable data about the shape and structure of the point cloud). So let's examine these algorithms...

## 3.1 Normal vectors

The point clouds are just a bunch of points in void that only makes an illusion of having a shape. Therefore the opengl itself has no way of soothsaying how to shade the points. The only way for the point clouds to be shaded is to provide a normal vector for each an every point in the cloud. When programming the project I went through some experiments of counting a mean normal vector of triangles made from nearest neighbours but the results were not very good-looking and took quite a time to count. The algorithm that I ended up with uses principal component analysis (PCA, as described for example in [4]) - first it gets all neighbours in a certain distance of the point thats normal it is counting and then constructs an oriented bounding box that is the smallest oriented box containing all the neighbour points. Seeing that the box is quite thin, as the points are planar-like, it is evident that the smallest side is perpendicular (or almost perpendicular) to the virtual plane inset between the points - this means it is the sought normal vector. The problem then is how to construct the oriented bounding box. The PCA helps with the answer. On the given set of points in

3D it returns 3 orthogonal vectors in the directions of the 3 biggest variances of the data. It works as follows (cycle through all the points in the point cloud):



**A pseudocode of the algorithm (for each point in the point cloud):**

1. FindNearestNeighboursInDistance(distance);
   IF(foundNeighbors30) THEN FindNNearestNeighbours(30);
2. CountCenterOfFoundNeighbors();
3. cov=CountCovarianceMatrix();
4. CountEigenValues(cov, l1, l2, l3);
5. Let l3 be the smallest eigenvalue: resultNormal=CountEigenVector(cov, l3);

Explanatory notes to the pseudo-code:

1. First it needs to get the data which the principal component analysis should be performed on. The algorithm first seeks for all the neighbours of the point thats normal is counted that are within a given distance. In case there aren't enough neighbours (with small numbers the algorithm doesn't work properly), I delete the result and find n nearest neighbours where n is at least 30 (this is the number that I found was the absolute minimum for the algorithm to give good results).
2. Then the mean in every coordinate is calculated. This is done in a simple for-cycle, nothing that needs further mentioning.
3. After we calculate the mean, the covariance matrix needs to be counted. This is done using the following formula:

$$cov_{i,j} = \sum_{k} \left(neighbours\left[k\right]_i - center_i\right) \times \left(neighbours\left[k\right]_j - center_j\right)$$

   where $cov_{i,j}$ represents the element of the matrix cov in its i-th row and j-th column. Notice that the cov matrix is a diagonal matrix (which by the way

means, that I didn't have to check the dictionary, for I always confuse rows and columns...).

4. Then I have to find the eigenvalues of the matrix. This is done using the linear algebraic formula for getting the characteristic polynomial of a matrix and then solving the roots of a cubic polynomial, which are the eigenvalues. If $cov = \begin{pmatrix} a & b & c \\ b & e & f \\ c & f & g \end{pmatrix}$, then the characteristic polynomial is

$$P(\lambda) = \begin{vmatrix} a-\lambda & b & c \\ b & e-\lambda & f \\ c & f & g-\lambda \end{vmatrix}$$

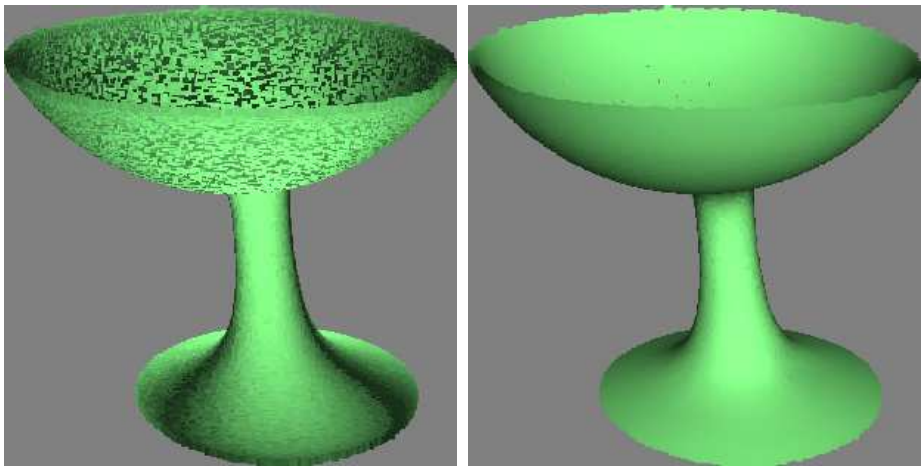
5. Once I have the eigenvalues counted (lets mark them as $(\lambda_1, \lambda_2, \lambda_3)$), I have to find the eigenvectors. The eigenvectors then are the vectors of the directions of the sides of the oriented bounding box. The eigenvectors are counted by solving the inhomogeneous linear equations system

$$\begin{pmatrix} a-\lambda_{1,2,3} & b & c \\ b & e-\lambda_{1,2,3} & f \\ c & f & g-\lambda_{1,2,3} \end{pmatrix} \times \begin{pmatrix} vx_{1,2,3} \\ vy_{1,2,3} \\ vz_{1,2,3} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

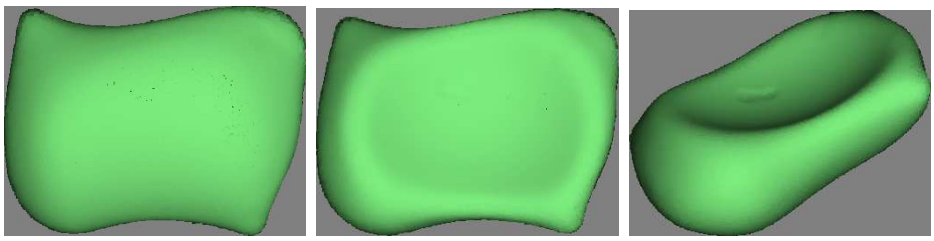
6. The calculation is almost over. When the eigenvalues of the covariance matrix are found and the eigenvectors they belong to are solved, the biggest eigenvalue's eigenvector holds the direction of the biggest variance of the points, the second greatest holds the second biggest variance, etc., which means, that the eigenvector belonging to the lowest eigenvalue of the covariance matrix holds the direction, in which the set of points has the smallest variance, i.e. the direction of the smallest side of the oriented bounding box, which is the normal vector we wanted.

### 3.1.1 Examples

Here are some examples of how the algorithm for counting normal vectors works:



The goblet was created from a sphere using the "potters wheel" and the antigravitation tool...



A pillow also shown in the level of detail section, ruck up by the antigravitation tool.

## 3.2 Level of Detail

The problem of the level of detail (or, more precisely, the problem of adding points so that the point clouds would appear to have continuous surface) wasn't really urgent, since the point clouds look quite good even when there are holes in them, but when the point clouds are deformed too much, they tend to have some areas visibly more dense than others, which looks awful.

The first attempt on counting the level of detail, that I made when programming the editor, was using the point cloud's journal of changes applied to the cloud - this is a data structure described elsewhere in this document; only note, that the journal provides operations for reconstructing the point cloud by starting at the basic shape

it was when inserted and then processing all the modifications that were performed. Since the number of points of the cloud is set when the cloud is created, I could simply create an almost equal cloud, that only had a greater (smaller) number of points. However, I rejected this method even almost before it was implemented, for it is very time consuming and doesn't solve the problem of various densities of the points throughout the point cloud's surface.

Seeing the fiasco of recounting the whole point cloud with a different number of points I had to find a way to insert new points dynamicaly only into those parts of the point cloud, that were thin and so new points were needed there. I took the liberty of trying to invent my own algorithm instead of trying to find one in literature. The main problem was to find the places where the points were too scarce and then insert new points into the right place. I based the algorithm on an idea similar to the one used when counting the normal vectors.

The algorithm goes through all the already present points in the point cloud and checks their surroundings. When checking the points surroundings, the algorithm first gets all the neighbours of the point in a given distance. When the distance is small enough, the neighbor points form a shape quite similar to a plane (I call this the "pseudo plane"). This part can reminds the reader of counting the normal vectors but since the algorithm presumes that the currently stored normal vectors are correct, the normal vector of the inspected point is used for getting new coordinates (x', y' and z') in which the "pseudo-plane" would lie in the x' y' plane and the z' coordinates would vary as little as possible.

Then in the x' y' plane a grid is constructed that splits the plane into fields (the grid has a previously defined number of fields) and for each field the number of points that lie within it is counted. If a field of the grid contains no points, it means that a point should be inserted into that field. That is easy - just compute the x' and y' coordinates of the field and place a point randomly inside it. If, however, the algorithm ended in this stage, the resulting point cloud would have bad-looking artifacts all around it and would by no means look smooth.

To help with this problem, while the grid holding the numbers of points is constructed, I construct another grid that holds the mean z' values of the points inside it. When a point needs to be inserted, its z' coordinate is counted as a weight mean of the other points' coordinates. In the end, the x', y' and z' coordinates are transformed back into Cartesian coordinates system.

This is the pseudo-code of the algorithm:

1. Foreach point in the cloud
    (a) FindNearestNeighboursInDistance(distance);

(b) CountPerpendicularVectors(vx, vy, vz=normal);

(c) NormalizeVectors(vx, vy, vz);

(d) convM = GetTransformationMatrix(vx, vy, vz);

(e) convInv = TransposeMatrix(convM);

(f) TransformCoordinates(neighbours, convM);

(g) ConstructGrids(t_Neighbours, grid, gridZ);

(h) Foreach emptyField in grid

i. CountNewXY();

ii. ApproximateZ(x, y, gridZ);

iii. InsertNewPoint(x, y, z);

(i) End

2. End

Note that the algorithm expects that all the normal vectors of the points are correct - i.e. no operation that changes the shape of the point cloud was performed on the cloud between the last recounting of normal vectors and the computing of the level of detail.

The algorithm first finds all neighbour points in a given distance from the examined point (the distance searched means the vicinity of each point that should be filled with points). Note that the points are browsed through the array but the neighbours are searched through the tree structure, in which the LoD points are inserted, so that the newly inserted points aren't inspected by the algorithm, only inserted.

Then the normal vector of the examined point is taken and two vector products are computed so that I receive two new vectors that are perpendicular to the normal vector and to each other. The two new vectors are then normalized (the normal vector already is).

Now comes the main idea of the algorithm. Since the point clouds have points only on surface, the small vicinity of each point forms something, that I call a "pseudo-plane". The points' coordinates in such a pseudo-plane vary only a little in the direction of the normal vector but rather a lot in the directions of the newly computed two vectors. Now since I have three normalized vectors, and the coordinates of all points are in the means of unit vectors in the direction of the x, y and z axes, the three orthonormal vectors can be written as three rows of a matrix that converts one coordinate system to another. (Plus, the inversion matrix is simply the conversion matrix transposed.)

When we've got the conversion matrix, the neighbourhood of the examined point is transformed into new coordinates (let's call them x', y' and z'). Tady bude vzorecek...

Since the depth of the object is very small, it has almost no effect on the density of the points - the density is distributed mainly in the x' and y' coordinates. So, seeing that all the points are distributed in the means of x' and y' I make a grid with a given number of rows and cols that has a certain size (dependent on distance searched for points in neighbourhood) in the means of x' and y' and count the number of points that would fall in the fields of the grid. Parallel to that, I have a grid storing the mean z' value of each field of the grid.

When both the grids are constructed (the counting and the z' grid), the grid holding the numbers of points is examined and if a field of the grid has zero points in it (or a too small number), the algorithm places a point randomly in the field (randomly in the means of x' and y'). If the algorithm was let in this state, it would work quite ok, only for that it would "blur" the surface of the cloud - the z' needs to be guessed before the point can be inserted.

The z' coordinate is counted using the second grid - using weight mean of the nearest values (in means of x' and y' coordinates) of the already present points' z' coordinate, I get the last coordinate needed, transform it back into the Cartesian system (this is done easily by multiplying the coordinate vector by the inverse conversion matrix) and insert it into the tree storing the points.

In the end, since all the newly inserted points are only in the tree-based data structure, they need to be transformed into the array used for drawing the points.

Better-looking results could be achieved by using Bézier surfaces, Fourier transformation or any other sophisticated interpolation method for achieving the z' coordinate of the newly inserted point.

This algorithm has the biggest advantage in its being very simple and little time-consuming (e.g. the level of detail operation took about 600 miliseconds on a 10 000 vertexes cloud on my computer).

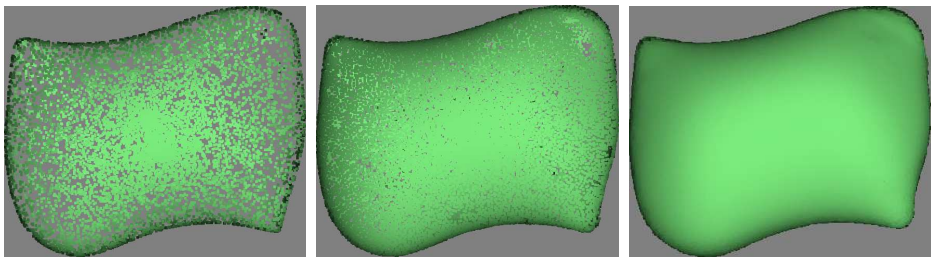
### 3.2.1 Insuficiencies and disadvantages to be improved

The main insufficiency that needs to be repaired is the relative crudeness of the algorithm for counting level of detail. It works rather efficiently when the points of the cloud aren't too thin but when the mean distance between the points of the cloud is orderly higher than the distance needed to keep continuous look, the algorithm tends to cluster the points and fill the empty space rather slowly. The solution (when using the same attitude for the algorithm) is to analyze the points of the cloud before inserting

the clouds using the grid and then make the insertion hierarchical - start with a wide grid and then reduce iteratively the size to the one desired while inserting points.

### 3.2.2 Example

Here is an example of the level of detail operation commited on a pillow created by pulling points from a sphere using the gravitation tool.



The first image shows the "pillow" just with correctly counted normal vectors. The second image shows the pillow with one iteration of LoD commited, the third image was commited LoD 3 times.

## 3.3 Deformation tools

The goal of the work was to implement an interactive editor and hence it should have some tools with which the point cloud can be edited. Apart from the compulsory tools for moving, scaling and rotating the point clouds, the editor contains also tools for pulling points from the point cloud, pushing into the cloud and cutting the points off the cloud.

### 3.3.1 Gravitation and antigravitation

When implementing the functions for pushing/pulling points (in the graphical user interface they are called gravitation/antigravitation and they shall be called so from now on) I took a big inspiration in physics. The gravitation/antigravitation deformation tools follow the idea of a center point, which is the source from which force is exerting (this is where the parallel to gravitation comes from) and all the points of the selected cloud are affected. The effect of the force depends mostly on the distance from the source of the force to the affected points. The rest of it is done by parametrization which is provided to the user through the gui.

The gravitation and antigravitation operations are pretty much the same - the only difference is that their "force" has opposite direction. The displacement vector indicating for each point the distance and direction the point should move because of the effects of the force affecting it is given by the equation

$$v = \frac{magnitude}{\left((px-cx)^2+(py-cy)^2+(pz-cz)^2\right)^{\frac{power}{2}}} \times \left(\begin{pmatrix} px \\ py \\ pz \end{pmatrix} - \begin{pmatrix} cx \\ cy \\ cz \end{pmatrix}\right)$$

where the vector p (px, py, pz) holds the position of the point and vector c (cx, cy, cz) holds the position of the source of the force. The magnitude parameter holds the value of the coefficient giving "how much the point should move independent of its position" - i.e. the volume of the force - the larger the value the more will the point move in the end. The power parameter describes the character of the force - how much distance affects the volume of the force - the bigger the value of the power parameter, the less will be the distant points affected.

### 3.3.2 Trimming the point cloud

The trimming tool provides the possibility to cut points off the point cloud. The main idea is that there is a "cutting plane", that the user controls and which divides the space into two semispaces. All the points in one semispace will remain as they are while all the points of the cloud in the other semispace will be ruthlessly deleted.

The cutting plane is represented using the normal vector of the plane and one point that lies within it. The points of the point cloud are then tested simply by making the scalar product of the normal vector of the cutting plane and the vector leading from the plane's point to the tested point. If the scalar product is greater than zero, it means that the point is to be deleted.

### 3.3.3 Usage of the deformation tools

The three described deformation tools are used in the cloud editor window of the PointCloud editor. The cloud editor window provides the "potter's wheel" functionality that makes the deformation tools the more interresting and extends their functionality. The "potter's wheel" functionality rotates the cloud in the gui (and in the inner representation rotates the tools around the cloud) and thus spreads the effect of the force evenly on all the points of the cloud - this helps with creating smooth shapes of the deformed clouds.

# Chapter 4

# Conclusion

I was able to propose and implement a standalone editor for 3D point-based graphics which provides to editing non uniform point clouds with considerable numbers of points. The editor provides four geometrical primitives to start with (a cube, sphere, cylinder and a rectangular plate) and several tools for modification. An interesting feature of the program is, that it is quite easy to switch the data structures used for searching in a points neighbourhood (these are used while counting normal vectors and level of detail) and thus the program provides the possibility of comparing different types and implementations of space partitioning data structures.

I'm rather satisfied with the results of the two main algorithms (the counting of normal vectors and level of detail) which are quite fast and provide good-looking results (of course the algorithms depend a bit on the input shape and number of points of the cloud, so there's still a lot to improve).

## 4.1 Future work

In the future I would like to add more deformation tools and geometrical primitives, possibly write a parser that could create point clouds from mathematical description of their shape. I would also like to try to supply the editor with data from a 3D scanner and examine its performance on real-life data.

I'd also like to continue on developing and improving the function for counting level of detail and make it possible to reduce the number points when the cloud is dense enough or far enough to look continuously. And I'd like to study ways of describing the "pseudo-surface" the points form for more precise insertion of new points.

Another interesting idea is to allow the user to add new points manualy - to "draw" new points and to provide the possibility for each point to have a different colour.

## 4.2 The end

To sum it, the PointClouds editor isn't perfect and I plan on developing it further, but I think (or rather hope...) that someday it could be used even for real-life tasks.

# Bibliography


[1] Markus Gross, Hanspeter Pfister: *Point-based Graphics*, Morgan Kauffman, 2007

[2] OpenGL ARB: *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, Addison-Wesley Pub Co.

[3] Bentley, J. L. *Multidimensional binary search trees used for associative searching.*, Commun. ACM 18, 9 (Sep. 1975),

[4] Jon Shlens, Tutorial on Principal Component Analysis. at http://www.cs.cmu.edu/ elaw/papers/pca.pdf

[5] The QT library online reference documentation at doc.trolltech.com

[6] OpenGL tutorials at nehe.gamedev.net

[7] The slides for computer graphics lectures from RNDr. Josef Pelikan at cgg.ms.mff.cuni.cz/ pepca

# Appendix A

# User's documentation

## A.1 About

PointClouds is above anything else my "Year's project" - a programming project at the Faculty of Mathematics and Physics which I have the luck to attend to. And last but not least, PointClouds is a 3D graphics editor for creating and editing point based graphics scenes.

## A.2 Installation

To install PointClouds you have to download and compile its source code. This may sound a bit weird and complicated, but it is not. First you need to obtain the source code. For that you may need to install an svn client on your computer. Then svn checkout svn://cgg.ms.mff.cuni.cz/PointClouds. This will download all the sources to your computer. Next you need to check if you have a development version of the QT libraries and if not, download them at www.trolltech.com. If you use a Unix-like system, you probably have C++ compilers installed, if you use Microsoft Windows, better download QT with the mingw compiler.

Now we can proceed to compiling the application and running it. Start your command-line tool, go to the directory where you have saved PointClouds, then move to .../Pointclouds/trunk/editor. Type qmake -project. This command creates a configuration file for the qmake utility to generate a Makefile. Its name will probably be editor.pro. Now edit the created file (editor.pro) and add this line under its last line: QT+=opengl. Save the file and type qmake. This command generates a Makefile and now you're ready for compilation. Type make. The compilation will take about 1-2 minutes to finish, so go

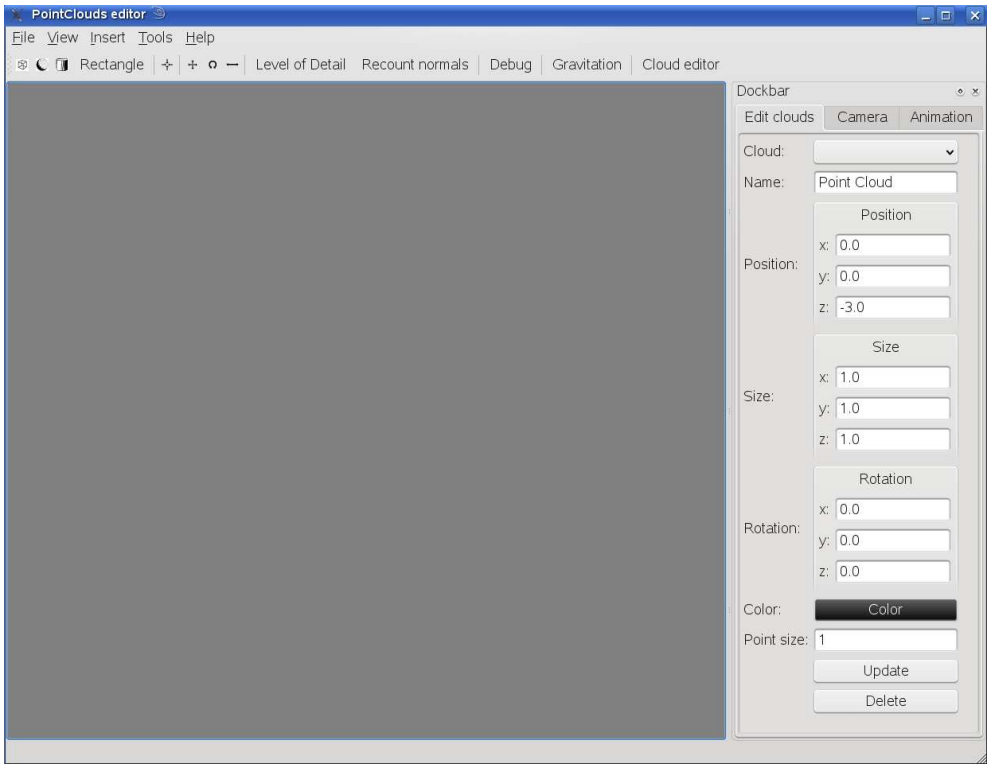



Figure A.1: The main screen of the editor

and have a cup of coffee :-)

To run PointClouds, simply execute its binary (editor.exe on Windows systems, editor on Unix-like systems).

## A.3 How to use PointClouds

When you run the application for the first time (and also any other...), it should look something like Figure A.1.

The big grey area covering most of the picture is the gl window - this will show you the scene and draw all the stuff. To the right of it is the dockbar - it holds information about the point cloud you have currently selected - its name, position, size, colour, etc. At the top are the toolbar and the menu, which will help you control the program.

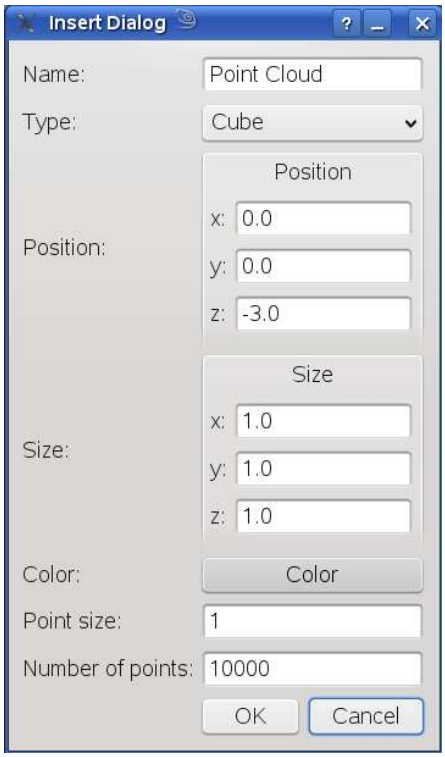



Figure A.2: The insert dialog

## A.4 Inserting the clouds

Since PointClouds is an editor and you have to have something you can edit, first of all you need to insert a point cloud. This is done either by clicking on one of the icons of primitives on the toolbar or by selecting the Insert menu, where you can select which cloud to insert.

Current options of point clouds primitives that you can insert are: a cube, a sphere, a cylinder or a rectangle. Notice that when you select a button for inserting one of these clouds, you don't get any options... it just creates a default shape. If you want to insert other than default point cloud of a selected shape, go to Insert $\rightarrow$ $Insertusingdialog$.

A dialog will pop up at you (Figure A.2). In the first column, you can type the name of the new cloud (this is just for your comfort, to better distinguish one point cloud from other). Then you can choose the type of primitive the point cloud will be. The options are a cube, a sphere, a cylinder or a rectangle. With time, there will come some more primitives.

Next you can specify the position and size of the new cloud. This is done by putting

absolute values of the x, y and z coordinates. By modifying the size you can distort the sizes and for example change a sphere into an ellipsoid.

One field further you can pick up the colour of the point cloud, then choose the point size. This means the pixel size of each drawn point. Typical values of the point size are 1-5... when you choose a bigger number, the resulting image will probably not look good - the large points can make it appear blocky.

The last column specifies the initial number of points in the cloud. The bigger the value the more will the resulting point cloud look like surface instead of a bunch of points. However, it also makes it more processor-consuming and on a normal computer if you select a value higher than 1 000 000, you'll end waiting a very long time for almost every operation you execute. So much for inserting the clouds, let's head to working with them.

## A.5 Working with the clouds

To work with a point cloud, you need to select it first. This is done either by selecting it in the list of the dockbar or by clicking the select button -  . When you select a point cloud, it gets lighter. In the main window of the PointClouds program you can do only a few basic modifications - move the clouds, rotate them, scale them (and delete them...). To do this, click the appropriate buttons (move -  , rotate -  , scale -  ), move the mouse over the gl window part of the program, click the mouse and move it. When you click the right mouse button, you can work with the other axis - with the left mouse button, you move the cloud in the left-right direction and the up-down direction, with the right mouse button, you move it in forward-backward direction, the same goes for scaling. When you are rotating the cloud, right mouse button rotates clockwise and counterclockwise from your view and left mouse button rotates in the remaining two directions.

You can also manipulate the selected point cloud by changing appropriate values in the dockbar. At the top of the dockbar you can select point clouds, then change their names, etc. just like in the insertion dialog except for that you can also rotate them here by typing numbers into the rotation columns.. To apply changes to the point cloud, simply click the Update button. You can also delete point clouds by clicking the Delete button, which is the lowest item of the dockbar.

Most of the operations that can be commited on the point clouds alter the shape of the cloud. For the point cloud to look as well shaded as possible don't forget to click the Recount normals button (this executes the shading computation).

You could also desire the point cloud to look more like continuous surface. This can be performed by clicking the Level of Detail button. Don't forget however, that the operation works properly only with correctly counted normal vectors, so the Recount normals button usually needs to be clicked before the Level of Detail button.

## A.6 Editing the clouds

To apply more sophisticated changes on the point cloud, you must first select it (first click on the select button and then click on the desired point cloud; if everything goes ok, the cloud changes its color to a lighter shade), and then click on the Cloud editor button on the toolbar or select Tools → $CloudEditor.Thatbringsupascreennotunlike FigureA.3$

There are three gl windows in the cloud editor. The main window is the biggest one, the top smaller gl window is the side view and the bottom smaller gl window is the look from the top. As for editing the point cloud, they are equal.

The odd looking blue-red-white triangle in the middle of the main gl window (right of the side gl window and in the lower part of the top view gl window) designates the source point from which the tools are taking effect. To move it left-right and up-down, simply press the right mouse button and move your mouse. If you want to move it forward/backward, use the mouse wheel.

## A.7 Camera

You may have noticed, that there are two groups of buttons on the right side of the window - tools and camera controls. We'll forget about the editing tools for now and look at the camera. If you click the Freelook button, move to one of the gl windows and holding the left mouse button, move the mouse - the edited point cloud will rotate allowing you to apply transformations from any side. Beware, the editing is not functionate while having the Freelook on, so click the button once more to terminate it.

If you want to use something like the "potter's wheel", as I call it, in the camera section, select one of the X/Y/Z buttons (this specifies the axis which the cloud will rotate around) and click the rotate button. The point cloud begins spinning around the axis you have selected. To stop the rotation, simply click the rotate button again.


Figure A.4: The tools section of the cloud editor window

## A.8 Editing tools

And now, onwards to the editing itself. By now there are three editing tools available (this will change since I have some more tools in my mind) - gravitation, antigravitation and trimming.

The gravitation tool is for pulling the points toward the operating triangle. Be careful, if you use the gravitation tool too long, all the points can end in just one point and you'll never get them back again.

The antigravitation tool is the precise opposite of the gravitation tool - it pushes the points from the operating triangle. The trimming tool deletes points in one direction from a position. We'll focus on this later.

Now let's explain more precisely how to use the editing tools. When you click the gravitation or the antigravitation button, a groupbox with properties of the editing tool comes up (Figure A.4). The properties box is identical for gravitation and antigravitation. The Power field specifies how much distance from the operating triangle to the points affects the distance the points will move (for mathematicians - the amount of force applied to the point is divided by the distance to the operating triangle powered by the value in the Power field). If you change these values, don't forget to click the Update button for the changes to take effect. Now that you have applied all the modifications you did to power and magnitude of the editing tool, move your mouse to one of the gl windows, click the left mouse button and see what happens. Note that the gravitation and antigravitation editing tools both edit the cloud in real-time - the longer you hold the left mouse button, the more transformation will be applied. Also note that when you hold the left mouse button, you can move the operating triangle.

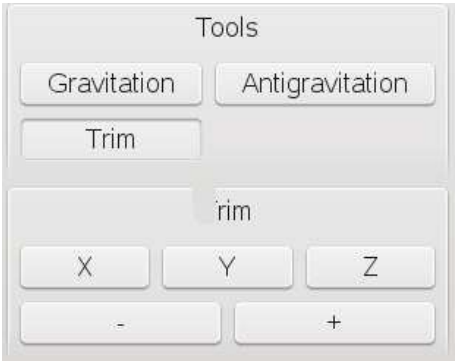



Figure A.5: Another view of the tools section of the cloud editor window

When you select the trim option, you get the possibility to select in which axis you want to trim (the X/Y/Z buttons) and which way in the direction of the axis to trim (this is done by the +/- buttons). The operating triangle changes to a rectangle with a line sticking out of it. The rectangle symbolizes the plane that determines which points will be deleted and the line sticking out of it means the direction in which the point are going to be eradicated. Points in the other direction remain the same as they were all the time.

## A.9 The remaining stuff...

Of course, you can save the scenes to files, open them, create new scenes, just as you are used to from other applications.

I hope you'll enjoy using the PointClouds program.
Yours sincerely
Vaclav Remes, author.

# Appendix B

# Programmer's documentation

## B.1 Brief characterization

Point clouds editor is a program for creating and modifying 3D scenes in point based graphics. As the name of the program indicates, the main subject are "point clouds" - an unorthodox representation of a 3D object made only by a bunch of points in the void that make an illusion of a continuous surface when displayed in a large quantity.

### B.1.1 Main issues

So, the main issues were - how to make an efficient representation of the point clouds in the memory, how to make them look good when displayed (this means counting the normal vectors and level of detail) and to choose a means of saving the scene to a file.

### B.1.2 Programming languages used

As a programming language I chose C++ for I like it the most and since I work in GNU/Linux and wanted the program to be natively created for this operating system, I chose the QT library for making the graphical user interface. For displaying the scene in 3D I used OpenGL. Thanks to the QT library being even for Windows, the program can be run in this (and some other) family of operating systems.

While working on Point clouds editor I tried to follow the basic rules of object-oriented programming and hence most of the code is encapsulated in classes. The only exception are mathematical functions, which are out of classes, stored in files math.c and math.h.

## B.2 Graphical user interface

### B.2.1 The main window - class CMyWindow

The main window of the program is taken care of by the class CMyWindow, which inherits from QMainWindow. It consists of the opengl window displaying the scene, the toolbar, main menu and a dockbar. The opengl window will be discussed later.

Most of the functions of this class have a rather intuitive purpose, so I'll describe only how to extend the functionality of the gui. When a new button(action) is inserted into the window, it needs a QAction object holding information about the button and its working function needs to be declared as a slot and then connected with the action (see the InitActions function for examples). The QAction object needs to be inserted either in the toolbar or main menu. For more documentation on programming the gui, read QT tutorials on www.trolltech.com.

The currently selected cloud's index is held in the selectedCloud variable, which can be used to index in CGLPCContainer to get the pointer to the selected cloud. If there is no cloud currently selected, the variable's value is -1.

### B.2.2 The opengl widget of the main window

The opengl widget is powered by the class CMyGLWidget, which inherits from QGLWidget. This class takes care of displaying the scene (all the clouds and light), walking through the scene, the camera effects and the basic realtime functions for manipulating the clouds (moving, scaling and rotating). Also when the user wants to select a cloud for further working, it is done here.

The main object of CMyGLWidget is a pointer to an object of class CGLCloudContainer called Scene, which, no surprise, contains all the point clouds displayed. A QTimer object called timer makes the widget redraw each and every 30 ms.

The most interesting functions of this class are the functions taking care of mouse input, so let's examine them further. Anything, that should be done while the user plays games with the mouse pointer over this window is designated by boolean values of mouseRot, mouseMove, mouseSelect, mouseMoveCloud, mouseScaleCloud, mouseRotateCloud and mouseGravitation. Values of these bools indicates if the camera should be rotated/moved, if the user is currently selecting a cloud in the scene and if the selected(if there is one) cloud should be moved, scaled, rotated or should be deformed by gravitation from other clouds.

All the previously described features are carried out in functions mousePressEvent, mouseMoveEvent and mouseReleaseEvent.

### Which object was clicked?

The mousePressEvent function is responsible of the actions done by a single mouse click. These are gravitation and selecting of clouds. The gravitation is a single operation performed on a cloud clicked by mouse. This relates with selecting the cloud, which is done by function getMouseSelectCloud(x, y), that gets the x and y coordinates and returns the index of the cloud displayed on position (x, y) or -1 if there is none. Getting the cloud is done in the program by choosing the select mode in opengl and drawing the scene only in the memory. The opengl functions for selecting object actually return a "name" - an integer identifying the drawn object - of the object being pointed on the position, but since I draw the clouds with their "names" set to their index in the clouds container (this will be discussed later), I can have the -1 for no object clicked and the other numbers I know will be only indexes of the clouds.

The same function (getMouseSelectCloud) is used when selecting the cloud for the user interface. But instead of calling a function on the clicked cloud, I use the QT library to emit a signal that carries the index of the cloud, so that other classes and functions can react on this signal separately from the code of this class.

### Moving, scaling, rotating

The basic real-time (most the time) operations that are possible to apply thanks to the mouseMoveEvent function. Since there is no delta function in QMouseEvent which would give me the difference to the last mouse position (and if there were any, I wouldn't trust them either), I have to remember the last known (or used for the computation) position of the mouse pointer - the member variable lastPos, which I refresh in each mouse(...)Event function after its body is done.

That allows me to calculate the difference to the last position at the beginning of mouseMoveEvent function and because the point clouds have implemented functions for moving, scaling and rotating the cloud not only by setting the absolute value, but also by assigning the relative difference that the cloud should be moved/rotated/scaled.

After any of these functions is executed, I emit the updateCloud signal for other classes to register, that the cloud's values have changed (this is done mostly for the CMyDockWindow class, as it is responsible of displaying the point cloud's details).

### The scene

For opengl to work properly, we first need to set up the shading model, depth testing and light properties. This is done in the initializeGL function.

The main painting part of the CMyGLWidget is carried out in the paintGL function. This function calls the draw method of the CGLPCContainer, that does the drawing of all the clouds, but the program also provides free looking and moving through the scene, which is done in paintGL function directly before painting all the clouds.

The abstraction of a camera is driven by 6 floating point values - posX(Y, Z) and mouseRotX(Y, Z). The pos(...) variables store the position of the viewpoint and their values are taken to life by the glTranslatef function, which moves the whole scene and makes an illusion of moving the camera. The scene is then rotated.

### The dockbar

The dockbar's main purpose is to inform about the status of the point cloud and its main values and to change the most general ones and to grant some functionality of moving the camera through the scene. When I look at it, I can see that the encapsulating class CmyDockWindow is there for nothing more than to separate independent parts of code and make the code more transparent.

The class holds a tab widget which contains all the other controls. Apart from many labels and groupboxes, which are there only for the purpose of design, there are several textboxes (instances of class QLineEdit) and buttons.

### The point clouds tab

When any values are changed using the graphical user interface of the dockbar, the Update button needs to be clicked to apply the changes, which invokes the update slot function. The update function then goes through all the values of the line edits and converts them to the appropriate data types and values. When the values are parsed, they are checked against the previous values the point cloud already had and the operations appropriate only to the changed values are performed. The changes are done simply using the selected point cloud's moveTo(x, y, z), resize(sX, sY, sZ), rotateTo(rX, rY, rZ), resizePoints(Size) and setcolour(colour) functions and the functions for setting the clouds name in the CGLPCContainer class and setting an items text in the combo box.

The combo box leads me to the part of choosing the point clouds in the combo box widget. Since I display only strings in the combo box for selecting the point clouds, I have to have the cloud's names in the combo box sorted exactly the same way as in the CGLPCContainer to be able to decide which object belongs to the selected name from the combo box. The combo box (cbclouds) is filled in the flushClouds function, that simply goes through all the names stored in the container and inserts them into the cbclouds in the correct order.

When the index of the cbclouds is changed (that means another item in the combo box was selected), the changeCloud(int) function of the CMyDockWidget is invoked, that gets the index of the selected cloud as a parameter and loads all the information about the point cloud (using the get(...) methods) into the QLineEdit widgets.

The last important function of the point clouds tab of the dockbar that needs to be examined, is the function for deleting. When the Delete button is clicked, the delCloud function is called and the point cloud is deleted from the container. Simultaneously, the values of the cloud are removed from the combo box and another cloud is selected. If there is none, the combo box is left as it is.

### B.2.3 The cloud editor window

The cloud editor window's view is split using the QHBoxLayout into two parts - a part with the function panel and the part with the opengl windows. The further layout is made using QVBoxLayouts and QGridLayout.

#### The opengl windows displaying the edited cloud

The three opengl windows displaying the edited cloud are objects of class CEditGLWidget with names editGLWidget, sideEditGLWidget1 and sideEditGLWidget2. As the names prompt, the editGLWidget variable is the main one and sideEditGLWidgets are its siblings. This affects the calls of their constructors - the "normal" constructor creates a standalone editing opengl window but the constructor for side widgets marks them as brethren of the editGLWidget and thus they share the cloud and tools (note that I have to remember which constructor was used in order to use the destructor properly) and are not standalone.

#### The deformations

The graphical user interface of the cloud editor window then transfers the given parameters of the editing tool (its type, source, magnitude, power, direction, etc.) after clicking the update button into the properties (mostly boolean and double variables) of the editGlWidget. The CEdigGLWidget class then overrides the mousePressEvent, mouseReleaseEvent and mouseMoveEvent. The other most important feature of this class is the timer variable, which grants the real-time deformations functionality (the timer is set to 30 miliseconds).

When the mousePressEvent and mouseReleaseEvent functions are invoked, the only thing they do is that they set the boolean values m_left and m_right that indicate if

the corresponding mouse button is clicked or not. The main part then goes in the mouseMoveEvent and onTimer functions.

If the right mouse button is held, the tools triangle (or square, depending on which tool is used - if (anti)gravitation or trimming) can be moved left/right and up/down. In the forward/backward direction it is moved by the mouse wheel, which is operated by the wheelEvent function. The mouseMoveEvent works so that it remembers the last mouse position in the mX and mY variables and then counts the difference to the current position of the mouse pointer. It then adjusts the variables related to mouse control - these are (depending on the function selected in the tools panel) mX, mY and camX, camY - the freelook rotation of the camera.

#### The timer - the main editing part

The main editing function is the onTimer - a function run every 30 miliseconds. The onTimer function takes care of repainting the window (with updateGL function), rotating the point cloud, when the rotation is on and performing the deformations on the cloud if left mouse button is down and freelook is off. When deformations are done, depending on the value of the action variable, the onTimer function calls either the gravitation or the pressure or the trim function of the edited point cloud. The gravitation and pressure functions depend only on the position of the tool triangle and the magnitude/power set by the user but the trim function also needs a vector indicating which direction the point cloud should be trimmed in.

The trimming vector is counted using the initial direction (the user selects the axis in which the point cloud should be trimmed and the +/- direction, so I can easily construct the corresponding vector) and then the vector is rotated to match the rotation of the cloud. The trimming function is observed in the part covering the CGLPointCloud class.

## B.3 The point cloud itself

### B.3.1 Before we can advance to the CGLPointCloud class

#### CGLPCContainer

Before we advance to describing the class representing the point cloud itself, let's take a look at the tiny little class storing all the point clouds. The main storing variable is the double pointer (pointer to an array of pointers) CGLPointCloud ** clouds. The clouds variable is used as a gum field. Apart from the clouds themselves, the CGLPCContainer class also stores the names of the clouds for better identification.

Most important functions:

- Clouds are inserted using the addCloud function, which creates a default name if the point cloud was given none.
- If it is necessary to get a pointer of a point cloud (for example because it was selected in the dockbar's combo box), the getCloud function returns a pointer of a point cloud on the position given to it as parameter or null if the position is invalid.
- The clouds can be also deleted from the container. This is done by the delCloud function, which edits the variables so that it seems as if the point cloud wasn't present, but doesn't delete it - it only returns its pointer (for case the programmer wants to do some further magic with it).
- The main function of this class is probably the draw function. It cycles through all the clouds and calls their draw function to display them. In the future this function could be improved to test whether the currently drawn cloud is visible or not and decide if it is necessary to draw it.

Apart for storing and drawing the point clouds that the container has within itself, the CGLPCContainer class is also responsible for saving and loading the scene into/out of a file. That is done by calling the save and load functions.

The scene is saved into a plain text file - since this is no commercial program, even a program for academical purposes, I think that it's best if the saved file is saved in a human-understandable way. The save function writes out the current position in the preallocated array storing the clouds, the value of the counter variable (the counter variable is used for generating names for the point clouds which weren't named by the user). Then it makes a cycle through the stored point clouds and writes them down - every new point cloud in the file is prefaced by "begin" and ended by "end". After the begin word the name of the cloud is printed and then the saving function of the cloud is invoked, which does the rest. The format in which the clouds are saved will be discussed later.

Loading is done as a complete reverse of the saving process (all the values have unique position, so there is almost no parsing with the exception of testing validity of data - e.g. if there is really a number where a number should be).

### CCloudJournal

The last thing that needs to be observer before I can continue to the CGLPointcloud class. This is the CCloudJournal class, which remembers all the changes that have been applied to the point cloud. The reason for this class is that when the scene contains too

many points, it could be useful when only the journal of changes can be transferred, another reason is that some changes, that were made, can be undesirable and so can be removed from the journal and the point cloud can be recounted. The replaying of the journal can, however, be very time-consuming.

The journal itself is actually a vector of journal entries (objects of class CCloud-JournalEntry). The CCloudJournalEntry holds information about which operation was invoked on the point cloud (the information is held in an integer) and the parameters, which are stored in a vector of doubles (since all the parameters are of numeric types, I simply store everything as a double and then cast them as the types I need.

Most important functions:

- To fill the journal, the Insert function is called with the parameter of a new CCloudJournal entry made up from the parameters of the function that affects the point cloud.

- The most interesting functions of the CCloudJournal class are createCloud and replayJournal. These two functions relate closely with each other for their meaning is to execute on the point cloud all the operations that are stored in the journal. The differences are just, that the createCloud creates a whole new cloud and the replayJournal function takes an existing cloud, clears everything that is within it and restores it back from the journal.

- The replaying of the journal is done quite simply - each CCloudJournalEntry is taken and applied to the point cloud using the applyChangeToCloud. The applyChangeToCloud function then tests the type of the journal entry and calls the appropriate function on the point cloud.

### B.3.2 CGLPointCloud

Finally, I reached to the most important class of the whole project - the CGLPointCloud. This class (and its subclasses) covers everything needed to create point clouds, save them to a file, apply deformations on them, count the normal vectors for the point clouds to look fancy and to thicken them to look continuous.

The CGLPointCloud class is located in the files glpointcloud.cpp and glpointcloud.h. Apart from CGLPointCloud there can be found also the typedefs for glVertex, glNormal, glVector and glcolour. These four types represent the same thing - an array of doubles the size of 3 and I use them only to visibly point out the purpose that variables of this types were created for.

Main variables of the CGLPointCloud class:

- glVertex * vertices - a pointer to an array storing the position of all the points in the point cloud (without this array the point cloud couldn't be effectively drawn on the display)
- glNormal * normals - a pointer to an array of normal vectors belonging to the point clouds (without this variable, the displayed points couldn't be shaded - they would all have the same colour independent of the shape of the point cloud)
- CTree * tree - a pointer to the CTree based object which helps with searching through the point cloud (will be discussed later in a separate section)
- CCloudJournal * journal which is an object that remembers everything that was done to the point cloud and that is capable of recounting the point cloud from scratch just by following all the applied changes.

Apart from these quite crucial objects, there are several not so sophisticated but also rather needed variables:

- for storing the position - GLFloat x, y, z
- GLfloat size - remembers the size of all the points (since a point is an abstract object that has no volume nor surface itself, I have to allow the user to manually decide the number of pixels each point will be drawn on).
- type - holding which geometrical primitive the point cloud arose from (now it could be from a sphere, a cube, a cylinder or a rectangle)
- the int v, maxv holding the size of the vertex array (v stores the position of the last vertex, the maxv is the size of the whole array - it is stored separately for the case of deleting some of the points - e.g. in the level of detail function, if the algorithm decides that there are redundant points which aren't needed for the point cloud to look continuous).
- The last two member variables that I haven't yet described are QColour colour and glVertex center. The purpose of the first one is clear - to hold the colour of the point cloud (I have thought about adding functionality for every point to have separate colour but I decided not to do it yet... the possibility to paint all over the point cloud is one of the possible future improvements).

**How to create a point cloud?**

The CGLPointCloud class has a private constructor. The reason for this is, that the points of a point cloud are sealed within the private section of the class and cannot be accessed any other way than by the deformation functions. The advantage of this approach is that by using the classes static functions for creating the point cloud, the newly created cloud has a defined shape, the points are only on the surface of the cloud and since all the presented geometrical primitives are analytically well-known, I don't have to use the quite time consuming recountNormals function to get their normal vectors (I simply count the normals using analytical geometry). Besides, how would a generic point cloud look like? Just a random bunch of points is not acceptable, so it's quite natural to create the object as a point cloud geometrical primitive.

So, as I said, the class has a private constructor, which means you cannot create normal objects using the new operator (or by writing CGLPointCloud cloud;). The only way to create a point cloud out of the CGLPointCloud class is by calling one of these functions:

- static CGLPointCloud * createCubeCloud(int num=def_num_points)
- static CGLPointCloud * createSphereCloud(int num=def_num_points)
- static CGLPointCloud * createCylinderCloud(int num=def_num_points)
- static CGLPointCloud * createByType(int type, int num=def_num_points)

The meaning of all the methods is quite clear - they create a point cloud of the desired shape and return a pointer to it. The num parameter of the functions designates how many points the newly created point cloud should have. The createByType function takes an additional integer parameter type, which indicates the type of the created cloud (sphere, cube, cylinder), tests its value and then returns the value of the appropriate function.

Note that the functions for creating new point clouds don't get any information about the sizes of the newly created cloud, only the number of vertexes. The reason is, that for the easiness of creating, computing the initial normal vectors, etc., the point clouds are created as unit instances of the desired primitives - e.g. the cube has a side of 1, the sphere has the radius of 1, the cylinder has the height equal to 1 and its radius also.

I said that the normal vectors are counted analytically:

- For the cube there are only 6 normal vectors, each belonging to one of the sides

- The sphere, since it is a unit sphere, has the normal vector's coordinates equal to the coordinates of the points.
- And the cylinder is a combination of both - for both the bases there are two normal vectors and the remaining points have the z coordinate equal to 0 and the other two are equal to x and y coordinates of the point the normal belongs to.

### CGLPointCloud::CTree and its offspring

The abstract CTree class defines the interface that the data structures for searching through the points of the point cloud need to implement. Currently these are the CDebugT, CKDTree and CBucketKDTree. These classes store all the points, which is redundant to the glVertex * vertices and glNormal * normals arrays. This constraint allows me to quickly shuffle between various data structures and maybe in the future each cloud could have different searching structure so that each cloud can have the one that fits it the most. For better description of these data structures see my bachelor's thesis.

### Basic functions of the point cloud

The most essential functions that contain almost no interesting ideas are the function for moving, rotating and scaling the point cloud.

- **Moving the cloud** The functions for moving the cloud are the move and moveTo. The difference between them is, that the move function takes the x, y and z parameters and moves the point cloud the specified distance whereas the moveTo function moves the point cloud to the give position.
- **Scaling the cloud** The scale and scaleTo functions work accordingly to the move functions, but rather than moving the point cloud they scale it. The scaleTo function doesn't work with scaling using the mouse but when inserting the scale value using the dockbar. The scaling itself works by multiplying the coordinates values of the point cloud.
- **Rotating the cloud** The last two basic functions are the rotate and rotate to. Just like the scaleTo function, the rotateTo function works only if used by the dockbar. The rotate function gets 3 parameters carrying the values of rotation in all the three axes (the rotating is done one after another by course of first the x

axis, then the y axis and then the z axis). Here's the example of rotating in the z-axis (a simple use of linear algebra...):

$$\begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

The rotateTo function then just uses the last values of rotation, rotates back (using the minus values) and then rotates to the desired position.

### Possible deformations

By the time of writing this text the Point Clouds editor offers only four deformation applicable to the point clouds (with two of them being almost the same). These are gravitation (2 types), pressure (or antigravitation) and trimming.

- **Gravitation** The gravitation has two functions, each doing almost the same but having different parameters. The easier one is the function void gravitation(double cx, double cy, double cz, double magnitude, double power, double minimum=0) used in the cloud editor window. This function takes the cx, cy and cz coordinates of the source of gravitation, takes the magnitude and the power of the gravitation and applies the force to all the points in a for-cycle. The equation indicating how much a point will move is:

$$v = \frac{magnitude}{\left((px-cx)^2 + (py-cy)^2 + (pz-cz)^2\right)^{\frac{power}{2}}} \times \left(\begin{pmatrix} px \\ py \\ pz \end{pmatrix} - \begin{pmatrix} cx \\ cy \\ cz \end{pmatrix}\right)$$

where px, py and pz are the coordinates of the point and v is the vector of translation. The coordinates of the translation vector are then simply added to the coordinates of the point. Note that the translation is different for each point and has to be counted in every run of the for-cycle, so that the points don't move uniformly.

The second type of gravitation takes another point cloud as a parameter instead of the cx, cy and cz values. Then it counts its "weight" (by the number of points divided by a reasonable amount) by which the magnitude is multiplied. Then it uses the same procedure as the "normal" gravitation function.

- **Pressure** The pressure (or antigravitation, as it is called in the graphical user interface) is quite similar to the gravitation function. The only difference is that the resulting translation vector of the gravitation function points to the source point while the resulting translation vector of the pressure function points from the source point.

- **Trimming** The third and last deforming function is the void trim(double cx, double cy, double cz, double vx, double vy, double vz). Since this functions purpose is to delete all points that are in a given direction from a given point, the cx, cy and cz parameters are the x, y and z coordinates of the point, from which the points of the cloud are deleted and the vx, vy and vz parameters designate the x, y and z coordinates of the vector indicating which direction the points should be deleted in. The there is (as everywhere) a for-cycle going through all the points and testing if they lie in the desired direction. The test is done using a scalar product of the vector from the source point to the tested point and the vectore given by vx, vy and vz. If the scalar product is greater than zero, it means, that the point lies in the desired direction and should be deleted. The deleting is done simply by moving the last point of the vertices array at the position of the deleted point and then reducing the total number of points in the array by one. Note that the variable i of the for cycle ( for(int i=0; i¡v; i++) ) needs also to be reduced by one - if it wasn't, the for-cycle would skip the replaced point.

### Saving/loading to/from a file

Saving of the point cloud into a file is done by the function void save( QTextStream & out ), where the out parameter is the stream into which the point cloud should be saved. When you think about it, you can find that the point cloud doesn't really carry too much information - just the colour, size of the displayed points, number of vertexes, the vertexes themselves and their normal vectors (which can be recounted, so the information is redundant) and the journal of operations performed on the cloud. Seeing this and since the point cloud is saved into a plain text file, I simply write the colour, number of vertexes, the size of the displayed points and then, in a for-cycle, the vertexes themselves with their corresponding normal vectors. And after the for-cycle, the journal is written.

Although it belongs to another class, I'll describe the saving of the journal here. The journal writes the type of the point cloud, the initial number of points (which can be different from the currently saved number of points), the colour, and in a for-cycle, every journal entry. The journal entries are written in the format: the type of the entry (the operation), whitespace, and then all the parameters delimited by whitespaces. Each

journal entry is then "ended" by newline.

The loading from the file is reverse to the saving - all the values have given positions, so no parsing needs to be done except for converting the numbers from string to doubles/ints and testing whether the input is correct.

### Remaining functions

The remaining functions are those necessary for everyday life with the instances of the CGLPointCloud class.

- resizePoints(GLfloat _size) - a function that sets the size of the displayed points
- setcolour - a function setting the colour of the whole point cloud
- getX, getY, getZ - functions returning the x, y and z coordinates of the point cloud
- getSizeX, getSizeY and getSizeZ functions are now deprecated and were used to getting the sizes in the x, y and z axes.
- So are the float getRotX, getRotY, getRotZ functions, that were used earlier when the rotations were done using the opengl glRotatef functions (now the rotating is cumulative and as such aren't stored).
- getcolour function is used to return the colour of the point cloud (only the value - it cannot be modified this way).
- getPointSize returns the size of the displayed points

## B.4 Conclusion of the documentation

I made the documentation rather brief for I believe that the code speaks for itself the best. I tried to describe the parts that I believe are most important in the code and I hope that any programmer who reads this documentation will be able to modify the program as easily as possible.

I am glad that the program as it is now can be used as an introduction into point-based graphics and its editing. I learnt a lot while programming it - from the QT library to computer graphics. The thing I am most proud about the editor is that my algorithm for computing level of detail works quite fast and gives unexpectedly good results. Of course it isn't ideal and I plan on working on it hard in the future.

What needs to be done the most on the project is extending the CGLPointCloud class with further deformation tools and testing more algorithms for the level of detail and normal computing.

My biggest dream (and goal) as the developer of this program is to see the program reaching a stable version in which it would be applicable even for other than academical purposes.

The program will be released as open-source and I hope that someone will be interested in it enough that they would like to take part in its development.